ML data abstractions, Operations on Types

CS152

Chris Pollett

Nov. 19, 2008.

Outline

- Finish up References and Recursive Datatypes
- Type Equivalence
- Type Conversion
- Control Structures

- As we talk about the above we'll continue to introduce ML.

Pointers and References

- We went over the last two slides on Monday quickly so I am briefly revisiting them.
- One type which does not correspond to a set operation is the pointer or reference type.
- This types corresponds to the set of all addresses that refer to a specified type.
- In C we could declare such a type using a syntax like: typedef int* IntPtr;
- To create a reference in ML we can do things like: val x = ref v;
- To create an actual reference type we could do:

  datatype ref_int = ref of int;

  We can modify a ref the value of a variable using :=

  We can get the value of a ref variable with !

Recursive Datatypes

- ML allows one to build up datatypes recursively:

```
datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
```

- One can then define functions on these recursive types.

Functions Using References

- We can use functions and references to do message passing in ML in a similar fashion to how we did it in Scheme:

```
datatype message = GetBalance | Withdraw of int;
fun make_atm data GetBalance = !data:int
    | make_atm data (Withdraw x) =
    (       data := !data - x;
            !data) (* parentheses are like begin/end in Scheme *)
val b = make_atm (ref 100);
b GetMessage;
b (Withdraw 10);
```

- make_atm is of type fn : int ref -> message -> int. Recall from last day, that we said that via **currying** such a type was roughly the same as fn : int ref * message -> int
- Note: the return type is an int. This might be awkward for some kinds of messages. To get around this we could create a datatype responses (like messages) for handling the types of each of the responses.

Functions Using Recursive Datatypes

- As we saw on the last slide, we can define functions for complex datatype by providing one pattern for each constructor of the datatype.
- So for our btree type we could define a function:

```
fun sum(Empty) = 0
| sum(Node(a, left, right)) = a + sum(left) +
    sum(right);
```

- This would be a function of type: int*int btree -> int

Type Equivalence

- Type equivalence is the problem of determining whether two types are the same.
- There are two main approach used by programming languages to do this: (1) use **structural equivalence**, (2) use **name equivalence**.
- Two types are structurally equivalent if they are built of out base types in the same way.
- For instance, if I defined by c and d to be int*char, then they would be structurally equivalent type. However, neither would be equivalent to the type char * int;
- Two items have name equivalent types if the names of their types are the same. So if x was of type c above and y was of type d, they would not be of name equivalent types.
- Most languages use a mixture of name and structural equivalence in determining if two items are of the same type.

Type Conversion

- Sometimes we have the need to convert from one type to a different type.
- Conversions might be implicit/explicit, where implicit conversions are called coercions. For example in C:

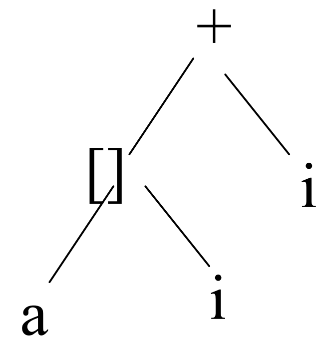
  1.0 * x/2 /* 2 is coerced to a float */

  (char)65 /* the int 65 is explicitly converted to a char */
- (char) is called a **cast**.
- If you convert from a bigger type to a smaller type it is called a **narrowing**. (like char example above).
- The opposite kind of conversion is called a **widening.**
- Different languages take different approaches to how often the programmer needs to explicitly convert types.

Type Checking

- Type checking is the process a translator goes through to verify that all constructs in a program make sense in terms of its constants, variables, procedures, and other entities.
- Type checking can be either **dynamic** or **static** depending on whether it occurs at run-time or not.
- An essential part of type checking is called **type inference**. This is where the types of an expression are inferred from the types of its sub-expressions.
- Given the typing of two sub-expressions of an expression, one needs to check if the operation that is being applied two subexpression makes sense in terms of their types. This is called a **type compatibility** check.
- In an **assignment compatibility** check of e1 = e2; the left hand side value (an **l-value**) must a reference to a place to store the right hand side value (r-value).

Type Inference



- We next consider the process of finding the most general types of the items in an expression based on the use those items.
- Consider the syntax tree of a[i]+i.
- A type checker would look up the types of each of the leaf items and percolate up a type for the internal nodes.
- Suppose we have the type of i as int. Are we forced on the types of the rest of the tree? Yes.
- In general, you might do a traversal of the tree, always labeling the nodes with the most general type.
- As we do this we might need to check that the type given to a node by its subtrees will match with the type we are expecting for that node.
- This is called **unification** and it might result in types in the node and the types in its sub-trees becoming narrower.
- If the type of a node changed then we retype the subtree of the weaker type.

Three Conditions for Type Unification

- Any type variable unifies with any type expression
- Any two type constants (that is, things like int or char) unify only if they are the same type.
- Any two type constructions (array, struct, recursive types) unify only if they are applications of the same type constructor and all of their component types also unify.